LSU EE 4755 Homework 4 Due: 28 October 2020

☣ *Paper copies will not be accepted. E-mail your solution to* `koppel@ece.lsu.edu`. *A single PDF file is preferred.*

This assignment refers to the solution to Homework 3. Pieces are shown below, the complete solution can be found at `https://www.ece.lsu.edu/koppel/v/2020/hw03-sol.v.html` *and in the directory where the original assignment was copied from.*

Problem 1: Using the simple model compute the cost and delay of the `nnAdd` module from Homework 3 (shown below) for both `sat=0` and `sat=1`. Do so after applying optimizations for constants. Show the cost and delay in terms of w. *Hint: See the simple model notes,* `https://www.ece.lsu.edu/koppel/v/2020/lsli-simple-model.pdf`*, for the cost of a ripple adder.*

- Show cost and delay in terms of w.
- Don't forget to optimize for constant values.
- Assume that the adder will be implemented using a ripple circuit.
- Indicate both the delay of the least-significant bit of the sum and the delay of the most significant bit of the sum. *Answering this part correctly and applying it to the other problems in this assignment will reveal something important about the impact of detecting overflow and of the different methods of doing so.*

```
module nnAdd #( int w = 5, sat = 0 )
              ( output uwire [w-1:0] so, input uwire [w-1:0] a, b );
   uwire [w:0] s = a + b;
   localparam logic [w-1:0] smax = ~w'(0);
   assign so = sat && s[w] ? smax : s[w-1:0];
endmodule
```

There are more problems on the next pages.

Problem 2: Using the simple model compute the cost and delay of the `nnMult` module from Homework 3 for `sat=1`. Let w denote the setting of both `wa` and `wb` (they are to be set to the same value), and let y denote the setting of `wp`. Solve this for $y < 2w$. Do so after applying optimizations for constants.

Solve this using the following cost for an unsigned integer multiplier with two w-bit inputs and a $2w$-bit output: the cost using the simple model is $10w^2\,\mathrm{u_c}$ and the delay is $[8w + 2]\,\mathrm{u_t}$ for the complete product and $[4i + 2]\,\mathrm{u_t}$ for bit position i. (The LSB is at position $i = 0$.) (For more details on how those were derived see the comments after the Linear Multiplier in `https://www.ece.lsu.edu/koppel/v/2020/mult-seq.v.html`.)

- Show the cost and delay in terms of w and y.
- Solve this for $y < 2w$.
- Don't forget to optimize for constant values.

```
module nnMult #( int wa = 5, wb = 6, wp = wa + wb, sat = 0 )
   ( output uwire [wp-1:0] p,  input uwire [wa-1:0] a, input uwire [wb-1:0] b );

   localparam logic [wp-1:0] pmax = ~wp'(0);
   localparam int wmx = wp > wa+wb ? wp : wa+wb;
   uwire [wmx-wp:0] phi;
   uwire [wp-1:0] plo;
   assign {phi,plo} = a * b;
   assign p = sat && wp < wa + wb && phi ? pmax : plo;

endmodule
```

There are more problems on the next pages.

Problem 3: Using the simple model determine the cost and performance of module `nn1xI` (shown on the next page) for the configurations described below. In all cases, let n denote the value of `ni`, w denote the value of `ww` and `wi` (which are the same) and y denote the value of `wo`. Assume the same hardware costs as the first two problems (modifying sizes and accounting for cascading where appropriate).

(a) Find the cost (not delay in this part) for `sat=0`, `tr=0`, and $y > 2w$ (that's one configuration) and for `sat=0`, `tr=1`, and $y > 2w$ (that's a second configuration). The two costs will be very similar.

- Show the costs in terms of n, w, and y.

(b) Find the delay (not cost in this part) for `sat=0`, `tr=0`, and $y > 2w$ (that's one configuration) and for `sat=0`, `tr=1`, and $y > 2w$ (that's a second configuration). The two delays will be very different.

- Show the delays in terms of n, w, and y.
- When computing the total delay don't forget to take into account the time that inputs arrive at each port, especially for the multiplier.
- When computing total delay account for cascading of ripple units.

(c) Find the delay for `sat=1`, `tr=0`, and $y > 2w$ (that's one configuration) and for `sat=1`, `tr=1`, and $y > 2w$ (that's a second configuration). The two delays should be very different from each other and from the delays from the previous problem.

```
module nn1xI #( int wo = 10, wi = 4, ww = 5, ni = 2, tr = 0, sat = 0 )
   ( output uwire [wo-1:0] ao,
     input uwire [wi-1:0] ai[ni],
     input uwire [ww-1:0] wht[ni] );

   if ( tr ) begin

      if ( ni == 1 ) begin

         nnMult #(wi,ww,wo,sat) mult(ao, ai[0], wht[0] );

      end else begin

         localparam int nlo = ni / 2;
         localparam int nhi = ni - nlo;
         uwire [wo-1:0] aolo, aohi;
         nn1xI #(wo,wi,ww,nlo,1,sat) nnlo(aolo, ai[0:nlo-1], wht[0:nlo-1]);
         nn1xI #(wo,wi,ww,nhi,1,sat) nnhi(aohi, ai[nlo:ni-1], wht[nlo:ni-1]);
         nnAdd #(wo,sat) add(ao,aolo,aohi);

      end

   end else begin

      uwire [wo-1:0] s[ni-1:-1];
      assign s[-1] = 0;
      assign ao = s[ni-1];

      for ( genvar i = 0;  i < ni;  i++ )
        nnMADD #(ww,wi,wo,sat) madd( s[i], wht[i], ai[i], s[i-1] );

   end

endmodule

module nnMADD #( int wa = 10, wb = 5, ws = wa + wb, sat = 0 )
   ( output uwire [ws-1:0] so,
     input uwire [wa-1:0] a, input uwire [wb-1:0] b, input uwire [ws-1:0] si);

   uwire [ws-1:0] p;
   nnMult #(wa,wb,ws,sat) mu(p, a, b);
   nnAdd #(ws,sat) ad(so, si, p);

endmodule
```

There are even more problems on the next pages.

Problem 4: Consider module `nnOxI` instantiated with `no=1`, `tr=0`, for both `sat=1` and `sat=2`. (A slightly simplified version appears below.) Let n denote the value of `ni`, w denote the value of `wi` and `ww` (which are the same), and let y denote the value of `wo`.

Assume that $2w < y < \lceil \lg n(2^w - 1)^2 \rceil$. That is, y is large enough so that the multipliers can't overflow but not so large that the adders can't overflow.

(a) Compute the cost and delay for both the `sat=1` and `sat=2` cases. For `sat=1` just re-use answers from the previous problems.

- Show answers in terms of n, w, and y.
- Don't forget that the value of `wo` in the `nn1xI` instantiations depends upon `sat`.

(b) In terms of the costs computed above is `sat=2` always better, always worse, or sometimes better than `sat=1`? Be specific of course.

```
module nnOxI #( int no = 4, ni = 2, wo = 10, wi = 4, ww = 5, tr = 0, sat = 0 )
   ( output uwire [wo-1:0] ao[no],
     input uwire [wi-1:0] ai[ni],   input uwire [ww-1:0] wht[no][ni] );

   // Compute number of bits to represent largest possible value that
   // can appear on an ao.
   localparam int wr = $clog2( ( 2**wi - 1 ) * ( 2**ww  - 1 ) * ni );

   if ( sat < 2 ) begin

      for ( genvar i = 0;  i < no;  i++ )
        nn1xI #(wo,wi,ww,ni,tr,sat) row( ao[i], ai, wht[i] );

   end else begin

      for ( genvar i = 0;  i < no;  i++ ) begin

         uwire [wr-1:0] ar;
         nn1xI #(wr,wi,ww,ni,tr,0) row( ar, ai, wht[i] );
         assign ao[i] = ar[wr-1:wo] ? ~wo'(0) : ar[wo-1:0];

      end

   end

endmodule
```

Problem 5: *Zero points will be given for the answer to this question, but please try your very best to answer it.* Suggest a method of saturating `ao` that avoids the extra `wo` bits needed (for `nn1xI`) when `sat=2` but also avoids the critical-path-killing saturation logic used when `sat=1`. Your solution could add extra ports to all modules except `nnOxI`. A correct solution would detect overflow under the same conditions as `nnOxI` does with `sat=1`.